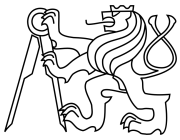
CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY



ASSIGNMENT OF MASTER'S THESIS

Title:	Dart implementation for Smalltalk/X
Student:	Bc. Branislav Havrila
Supervisor:	Ing. Marcel Hlopko
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of summer semester 2016/17

Instructions

The aim of the thesis is design and implementation of a compiler and runtime engine for programs written in the Dart language in Smalltalk/X.

1. Analyze the possibility of reusing the existing Smalltalk/X bytecode engine.
2. Describe the architecture of your solution and implement it.
3. Demonstrate the completeness of your solution on a set of examples using Dart Standard Library modules: core, collections, html, io, math, and mirrors.
4. Compare the performance with the Dart VM.

References

Will be provided by the supervisor.

L.S.

Ing. Michal Valenta, Ph.D.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague February 3, 2016

Czech Technical University in Prague
Faculty of Information Technology
Department of Software Engineering

Master's thesis

Dart implementation for Smalltalk/X

Bc. Branislav Havrila

Supervisor: Ing. Marcel Hlopko

9th January 2017

Acknowledgements

I would like to express my special thanks to my supervisor Ing. Marcel Hlopko for his exceptional advice as well as to Alexandra Antlová for her support and love. Secondly I would also like to thank to my parents and friends for their support during my studies.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on 9th January 2017 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Havrila, Branislav. *Dart implementation for Smalltalk/X.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2017.

Abstrakt

Táto práca sa zaoberá podporou programovacieho jazyka Dart v Smalltalk/X, porovnáva jazyky Dart a Smalltalk, prezentuje možnosti implementácie jednotlivých konštruktov jazyka Dart a taktiež porovnáva výkon kódu preloženého do bytekódu Smalltalk/X s Dart VM.

Klíčová slova Smalltalk, Dart, bytecode, AST, compiler, parser, třída, knihovna

Abstract

This thesis deals with the support of Dart programming language in the Smalltalk/X. It compares the Dart and Smalltalk languages, presents possibilities of implementing a support of Dart language features inside the Smalltalk/X and compares the performance of the implemented solution compiled into Smalltalk/X's bytecode with the Dart VM.

Keywords Smalltalk, Dart, bytecode, AST, compiler, parser, class, library

Contents

List of Figures

Introduction

Dart language has become a popular programming language over the past few years. Developers can use the Dart language for their web applications, server applications and now also mobile applications thanks to the Flutter project[1]. When building the server applications, developers can use the Dart Virtual Machine to run their scripts or programs.

The Virtual Machines have been here for a few decades now and Smalltalk has been one of the first programming languages to adopt the concept of a virtual machine. However, over the time, virtual machines have evolved a lot and there are more approaches how to evaluate the code. The implementation of Smalltalk/X uses compilation to bytecode as an intermediate representation and then the bytecode is interpreted or it might be compiled to native C code with JIT compiler if it is executed often enough.

Dart VM is not a bytecode based virtual machine and some reasoning about this can be found in the article "Why Not a Bytecode VM?" [2]. One of the main reasons why the authors of the Dart language decided to create a new virtual machine and not to make it a bytecode based is the development process. Skipping the compilation to bytecode step and executing the code directly should make the use of the Dart language pleasant and simple. The authors also make a reference to the Smalltalk live editing feature.

As the Smalltalk/X is using bytecode and has the live editing feature, this brought an idea of adding the support of Dart language into Smalltalk/X and comparing the performance of the Dart VM and Smalltalk/X.

Chapter 1

Problem statement

The purpose of this thesis is to create a parser and compiler for Dart language or at least its subset that would allow the Dart code to be executed in the Smalltalk/X. Therefore, the current Smalltalk/X has to be analysed how it would be possible to bring the functionality into the Smalltalk/X in a way, that we don't have to create a completely new bytecode based VM, but instead we use the Smalltalk/X VM to execute the code written in Dart language.

If the analysis brings a successful results, we can move to the implementation part and measure the performance of our solution afterwards.

1.1 Smalltalk/X

Smalltalk/X is a complete implementation of the Smalltalk programming language, written by Claus Gittinger[3]. Smalltalk/X compiles the Smalltalk source code into bytecode that is interpreted upon execution or it can be recompiled by the JIT compiler if the particular code is executed often enough. This might bring some performance advantages over the Dart VM.

As Smalltalk is a completely dynamic system allowing the classes, its fields and methods to be created, added, edited or removed while the system is running, the iterative development process is a very pleasant way to do the programming in Smalltalk. This brings the instant feedback and results as the developer doesn't have to go through the whole recompilation and run cycle.

1.2 Dart language

Dart is a class-based, object-oriented language. It is optionally typed, supports mixin-based inheritance and actor style concurency. [4]

It was initially designed by Lars Bak and Kasper Lund, developed at Google. It is a general purpose application programming language that is meant to be easy to learn, scale and should be deployable everywhere.

1.2.1 Important concepts

Types in Dart are based on interfaces and not on classes. This means that every class in Dart creates an implicit interface that other clasess can implement.

Dart has no final methods and almost all method can be overriden with a few exceptions like some built-in operators. However, this doesn't mean that Dart doesn't have final variables. These can be defined as well as final class fields.

Dart abstracts from object representation by ensuring that all access to state is mediated by accessor methods.[4] This means that whenever a field is accessed, it is done so through an accessor method, e.g. a getter method.

Dart supports top-level functions(e.g. the *main()* function) and also nested functions, i.e. a function defined in a function. Similarly, Dart supports top-level variables.

Unlike many other languages, Dart doesn't use *public*, *private*, *protected* keywords for defining the fields or methods. To define a private identifier(e.g. a field), it has to start with the underscore "_" sign. This identifier is then visible only within its library.

1.2.2 Types in Dart

Dart language is an optionally typed language and this should provide a balance between the advantages and disadvantages of types.

Types can be a source of documentation, they make the code more readable, they can help to find errors in the code by the static analysis and having types can improve the performance of the executed code as well.

On the other hand, complex type systems might get counter-productive if the developers end up with trying to satisfy the type checker instead of doing the productive work. Also, these advanced type systems are typically harder to learn and to work with.[4]

So Dart as optionally typed language offers programmers to choose whether they want to use types or treat the language as completely dynamic. However, the type annotations, that are added by a developer, help tools to do a better job in supporting the programmers, giving a better suggestions or warnings.

1.2.3 Classes

As mentioned in previous sections, Dart is a class-based language and it support the single inheritance of classes but extends it with the mixin-based inheritance. Superclass can be specified explicitly or can be left out, in which case the class inherits from the *Object* class. Therefore, every class has a superclass except for the *Object* class.

Classes can have fields defining the object's state and methods defining the object's behaviour. Apart from that, classes can also contain static methods and static variables(or class variables).

When defining fields, developers can choose to compute the actual value instead of storing it as an instance field. This can be done by defining the computed accessor methods. Syntactically, they are accessed in the same way as other fields, but when this computed field is accessed, the actual accessor method computing the value is called.

As Dart classes implicitly create an interface, explicit keyword for defining an interface does not exist in Dart. Interfaces are therefore defined by abstract classes. Checking if a class conforms to an interface at runtime is done through *is* keyword. The *is* keyword, however, has a major difference in functionality that it might have in other languages. It is actually checking if the object does conform to the interface rather than checking if the object is an instance of a given class.

1.2.4 Other language features

- Mixins: Mixins are a powerful way of reusing a class's code in multiple class hierarchies.[5] To use a mixin, the *with* keyword has to be used followed by one or more mixin names.

  To implement a mixin, developer has to create a class that declares no constructors and has no calls to super.

- Generics: A developer can use generics but as types are optional in Dart, he or she doesn't have to use them at all. However, it might help to improve the readability of the code as well as getting better IDE suggestions or better results from other tools that a programmer may use when using the static analysis. Also Dart VM supports the *checked mode* when the VM is checking if the object has a correct type and it can do so e.g. on the collections as well if the generics are used.

- Libraries: Libraries can be a powerful way to create a modular code that can be easily shared. Dart provides support for defining libraries using the *library* keyword and they can be imported in other files using the *import* keyword.

- Control flow statements: Dart has traditional control flow statements like *if* and *else*, *for* loop, *while* and *do-while* loops, *break* and *continue*, *switch* and *case*, and *assert*.

- Asynchrony support: Dart has several language features to support asynchronous execution. The most commonly used are *async* and *await* expressions.

- Metadata: Dart has a built-in support for adding metadata to the classes, fields or methods, known in other languages as annotations. For example, they can be used to mark method as a deprecated one or to create a todo note.

1.3 Smalltalk/X and Dart

Smalltalk was one of the languages that influenced the design of Dart and probably this made Dart a pure object-oriented language. We can benefit from the fact that everything is an object in the Dart language as well as in Smalltalk. Even primitive types like numbers or boolean values are objects in both languages.

Despite the fact that both languages are pure object-oriented languages, there are differences in how the classes behave. For example, if an object wants to access a field of another class in Smalltalk, it has to do through a message send, i.e. by calling the accessor methods. This Smalltalk encapsulation implementation is very similar to the implementation in Dart but there is a major difference. If an object wants to access its own field in Dart, an accessor method is always called. In Smalltalk, no accessor method has to be called if an object is accessing its own field. That said, Dart cannot override its getter or setter methods and it is a compile time error to redefine a getter or setter method of an already defined field. In Smalltalk it is completely fine because no default getter or setter methods are defined for the fields.

Dart has constructors for instantiating new objects whereas Smalltalk uses *new* and *basicNew* methods for allocating new objects usually followed by a call to some initialize method. Behaviour of the *new* and *basicNew* might slightly differ across different implementations of Smalltalk(e.g. new might call an initialize method). Dart has also some more syntactic sugar when it comes to constructors and fields initialization and it has also support for named constructors.

A one of the many similarities between the languages is that there are no private, protected or public keywords. Every method in Smalltalk or Dart is public. As Dart has automatically generated getter and setter methods all the fields are public. However, there is one exception: field starting with an underscore "_" sign is considered as private and should be visible only within the current library.

When it comes to overriding methods or flexibility in general, Smalltalk might appear a bit more flexible as basically any method can be overridden by a subclass whereas some operators are not overridable in Dart and Smalltalk also allows any system class to be changed easily. On the other hand, Dart has some more support in the control flow statements, like *break*, *continue* or *switch* statements.

Both languages support exceptions. In Dart however an arbitrary object can be thrown(or raised) rather than just throwing a supported exception or error types in Smalltalk.

Another difference between the languages can be found in the top-level functions and variables. Smalltalk doesn't support top-level functions nor variables and everything has to be defined within a class.

Chapter 2

Analysis

In this chapter we will focus on the options of using the Smalltalk/X bytecode when compiling the code written in Dart. We will analyse how we can reuse existing Smalltalk classes to run the Dart code.

2.1 Compilation of Smalltalk code

As we know that Smalltalk/X is compiling its classes to bytecode first and that's what we want to do with our code we can focus first on how Smalltalk/X is parsing and compiling the code written in Smalltalk.

2.1.1 Overview of the parse and compilation process

When a method or a change in a method source code is accepted in the Smalltalk editor, a *ByteCodeCompiler* class is the one responsible for compilation. *ByteCodeCompiler* is a subclass of the *Parser* class and *Parser* class is a subclass of the *Scanner* class. But before going into the *ByteCodeCompiler* code, there exists a *ClassDescription* class which decides first what class is its compiler class or in other words what class is responsible for compiling the code of the current class. This means that there might be a different compiler for each class, e.g. Smalltalk/X has already some support for Javascript compilation and the *ClassDescription* class for the Javascript classes return the *JavascriptCompiler* class.

When the correct compiler class is resolved by the *ClassDescription* class, the code then moves into the compiler. For Smalltalk classes, the *ByteCodeCompiler* class is used and therefore the execution continues here. The *ByteCodeCompiler* then asks its superclass which is the *Parser* to return an AST first by calling the *parseMethodBody* method.

Parser is continuously asking its superclass *Scanner* for next token and creates an AST which is afterwards returned to the *ByteCodeCompiler*. It processes the AST and generate an array with symbolic code. This symbolic

code is an intermediate representation of the code and is composed of Smalltalk symbols(e.g. *#pushInstVar1*) which is much easier to read and debug by a developer working on the compiler features.

After the generation of the array containing symbolic code the *ByteCodeCompiler* converts this symbolic code to the actual Smalltalk/X bytecode and is finally ready to create a new method and add it to the current class.

To summarize this process, after the method is accepted by the editor, the code has to go through these steps to generate a bytecode:

- *ClassDescription* obejct decides what compiler to use
- Resolved compiler asks *Parser* to generate an AST from the current code
- Parser continuously asks *Scanner* for next token and creates an AST sequence which is returned to the compiler
- Compiler generates a symbolic code from AST returned as an array
- Compiler converts symbolic code to the actual bytecode
- Compiler creates a new method object with this bytecode and passes the new method to the current class object by calling the *addSelector:withMethod:* method.

2.1.2 Understanding Smalltalk class model

Before we find out how to create a class in Smalltalk/X we have to understand its class model.
Object class is the superclass of all classes and it is the only class that has no superclass. Every class has its *class class* object in runtime and a *metaclass* object which is an instance of *Metaclass* class. *Metaclass* has also its *Metaclass* class instance.

Now there is an important thing to understand. When creating a new class, a new *Metaclass* instance is created for this class as well as the *class class* object. Interesting point is that *Object class* has no superclass but still responds to methods that are defined in *Behavior*, *ClassDescription* and *Class* classes. This behaviour should be set up in the VM initialization.

Assuming that we create a class named *MyClass* that has the Object class as its superclass the class inheritance for metaclass objects starting from MyClass moving up to the superclasses is following: *MyClass class* → *Object class* → *Class* → *ClassDescription* → *Behavior* → *Object*.

2.1.3 Class creation

Now that we know that there are Metaclass objects in Smalltalk, we can move to the class creation. When a new class is being added to the Smalltalk/X system, it is being done through a *Metaclass* instance. A new *Metaclass* object has to be created before the actual class is created. When a Metaclass object is created a new class can be created by calling *Metaclass*'s instance method *name: inEnvironment: subclassOf: instanceVariableNames: variable: words: pointers: classVariableNames: poolDictionaries: category: comment: changed: classInstanceVariableNames:*.

Important parameters in this method call are name, subclass, instanceVariableNames, category and classInstanceVariableNames.

- name parameter defines the name of a new class and it should begin with the namespace e.g. *MyNamespace::MyClass*
- subclass parameter defines a subclass of the newly created class
- instanceVariableNames parameter defines a list of instance variables. It is passed as a string with variable names separated by space
- classInstanceVariableNames parameter defines a list of class instance variables and it is also passed as a string with variable names separated by space
- category defines a category which the class belongs to

This *Metaclass*' method returns a new class that can be used in parse and compilation process when adding new methods to the class or calling class' methods.

2.1.4 Bytecode and symbolic code

As we are going to generate symbolic code and then a bytecode from the Dart source code we should be familiar with the symbolic code. Checking if we have generated the correct bytecode might be hard. Fortunately, there exists a *Decompiler* class which is a subclass of *ByteCodeCompiler* and it might be very useful as it can help us examine whether we've generated correct bytecode by decompiling our method back into symbolic code array. We can use this class also for decompiling methods written in Smalltalk to check what symbolic code we should generate in order to implement specific functionality.

Example method that just returns an instance field of the class written in Smalltalk looks like this:

```
myField
    ^myField
```

If we use the Decompiler for decompiling this Smalltalk method we get the following result:

```
1: 08 02      LINE [2]
3: A6         retInstVar1
```

We can see some debug informations on the first line. The *08* bytecode represents a line number bytecode and it is followed by one byte containing the actual line number value. Then the actual instruction is followed. In this case *A6* represents a bytecode for returning the first instance variable in the object.

2.1.4.1 Symbolic code overview

Smalltalk/X has currently more than 240 bytecodes and there's a symbolic code for each bytecode. Many of them are aimed for the optimization so we can generate bytecode that needs less instructions to be executed, e.g. there exists a bytecode for pushing first method argument on the stack. This saves a few instruction compared to the case where VM knows it has to push a method argument but has to read another bytecode to get the index of the method argument which should be pushed on the stack.

The most important bytecodes and symbolic code symbols for our needs are listed below:

Symbols used for pushing values on the top of the stack:

- *#pushNil* - pushes nil object on the stack
- *#pushTrue* - pushes true object on the stack
- *#pushFalse* - pushes false object on the stack
- *#pushLit* - pushes a literal value stored in the current method on the stack. It is followed by an index of the literal.
- *#pushSelf* - pushes self object on the stack, i.e. object whose method is currently being executed.
- *#pushMethodArg* - pushes a method argument on the stack. It is followed by an index of the argument.
- *#pushMethodVar* - pushes a method variable on the stack. It is followed by an index of the variable. Each method has a list of variables so it can allocate space needed to store the variables.
- *#pushBlockArg* - pushes a block argument defined in the current block on the stack. It is followed by an index of the argument.
- *#pushBlockVar* - pushes a block variable defined in the current block on the stack.

- *#pushInstVar* - pushes a current object's instance variable on the stack. It is followed by the index of an instance variable.
- *#pushOuterBlockArg* - pushes an argument on the stack that exists in the outer scope of the current block.
- *#pushOuterBlockVar* - pushes a variable on the stack that exists in the outer scope of the current block.
- *#pushGlobalS* - pushes a global symbol on the stack. e.g. a class name's symbol that might be followed by a message send - a call to a class instance method.

Symbols for storing values:

- *#storeMethodVar* - stores the current value on the top of the stack in a method variable. It is followed by the index of the method variable.
- *#storeBlockVar* - stores the current value on the top of the stack in a variable defined in a block. It is followed by the index of the block variable.
- *#storeInstVar* - stores the current value on the top of the stack in an instance variable of the current class. It is followed by the index of the instance variable.
- *#storeOuterBlockVar* - stores the current value on the top of the stack in a block variable defined outside of the current block. It is followed by the index of the outer block variable.
- *#storeClassInstVar* - stores the current value on the top of the stack in a class instance variable. It is followed by the index of the class instance variable.

Symbols used for returning values:

- *#retTop* - returns value on the top of the stack.
- *#retSelf* - pushes self on the stack and returns.
- *#retNil* - pushes nil on the stack and returns.
- *#retTrue* - pushes True object on the stack and returns.
- *#retFalse* - pushes False object on the stack and returns.

Symbols used for control flow:

- *#falseJump* - if top of the stack is the False object jump to the instruction specified on the next position of the symbolic code array
- *#trueJump* - if top of the stack is the True object jump to the instruction specified on the next position of the symbolic code array
- *#nilJump* - if top of the stack is the nil object jump to the instruction specified on the next position of the symbolic code array
- *#notNilJump* - if top of the stack is not a nil object jump to the instruction specified on the next position of the symbolic code array
- *#jump* - jump to the instruction specified on the next position of the symbolic code array

Symbols used for message sends / method calls:

- *#send* - sends a message to the object on the top of the stack. It is followed by a number of arguments and a selector symbol.
- *#sendSelf* - sends a message to the current self object. It is followed by a number of arguments and a selector symbol.
- *#sendDrop* - sends a message to the object on the top of the stack and pops the returned value from the stack. It is followed by a number of arguments and a selector symbol.
- *#superSend* - sends a message to the current self object requesting to execute a super implementation. It is followed by a number of arguments and a selector symbol.

These symbols should be enough for most of the cases in our Dart compilation. Now we have to focus on parsing the Dart code and its specific concepts that we have to consider in the parse and compilation process.

2.2 Parsing and compilation of Dart code

Dart is an open source project and its parser and compiler is available online as a part of the Dart SDK open source code. Therefore we may look at the code which makes it easier to implement the behaviour correctly.

Dart Parser and Compiler is a bit different from the Smalltalk way of compilation. First of all, Compiler is not a subclass of the Parser class, nor the Scanner class. The parsing process couldn't be done in one run. Dart Parser has to parse the top level first before parsing the actual functions. Scanner creates tokens from the source code while parsing the top level. One of the reasons why top level parsing has to be done first is that properties or

class members might be defined at different places within a class definition, whereas in the Smalltalk code, all the instance variables or class instance variables are defined in one place. Therefore Dart has to parse the top level first to be aware of what global variables or instance variables and classes are available and after that, the Parser can parse the actual function definitions.

When the parsing is done and Compiler gets the parsed function with AST, it continues doing optimizations like building a *Flow Graph* which is one the optimization techniques so the Dart code can run faster in the Dart VM. In our case the Compiler has to build the Smalltalk/X bytecode that will be executed.

2.2.1 Dart libraries

One specific thing of the Dart language which is not present in the Smalltalk/X are libraries. Dart library can contain top level functions, e.g. the *main()* function and also global variables. A library can be imported within other libraries using the *import* keyword and the classes, functions or any variables defined within this library can be used in the current library. Libraries are created by default so every app is a library even if it is not specified by the *library* keyword. Apart from the modularity that libraries create, they are a unit of privacy as all the identifiers that start with the underscore(_) sign are visible only inside the library where they were defined. [5]

Smalltalk/X has a concept of packages that contain classes but it's not possible to have variables or top level functions within these packages. These packages neither support any further privacy to the class fields or methods.

The concept of libraries can be easily simulated by creating equivalent library classes that contain class instance methods and class instance variables as a replacement for the global variables and top level methods but the privacy behaviour has to be either simulated by generating special bytecode sequence that checks the current library every time the method is called or a field is accessed or leaving this functionality out of the implementation and stating that this functionality is not supported in Smalltalk/X.

2.2.2 Types and type checking

As already mentioned, Dart is an optionally typed language so developers can completely omit type annotations when writing the code. However, types can be useful and if developer adds types to his or her code, Dart performs a static type checking providing better feedback and suggestions about the code to the developer.

The runtime checks are not performed by default. Nevertheless, Dart VM supports a special runtime mode called *strong mode*. If the *strong mode* is enabled, apart from the static type checks done by Compiler, Dart makes also runtime type checks that ensure the correct type e.g. when casting an object

to another type. When compared to Smalltalk, which is a dynamic language and does not do any type checks in runtime it would be very complicated to add this behaviour and therefore we should stick to the normal Dart mode where the type checks are not performed. Also omitting these runtime type checks is better from the performance point of view.

2.3 Compilation into Smalltalk bytecode

As there are no major impediments that would restrain compiling the Dart code into Smalltalk/X bytecode, we can start focusing on the details. Creating equivalent Smalltalk classes is the first step that has to be analysed.

2.3.1 Constructors

As Smalltalk has no constructors if we are not considering the *new* method, we have to introduce this concept here. I have chosen an approach where I create a class instance method for every constructor. This method contains only automatically generated code that creates a new instance of the class and calls an instance method on this newly created instance which corresponds to the constructor.
Let's consider a class called *MyClass* with a constructor that takes one parameter and it initializes one instance field *myField* with the value from this parameter.
This can be implemented in Dart with the following code:

```
MyClass(myParam) {
  myField = myParam;
}
```

When creating a new instance this results to the following Dart code: *new MyClass(myParam);*
This will be implemented in Smalltalk as a *class instance method* called *MyClass:* and initializer instance method called *MyClass:*. Class instance method with Smalltalk code will look like this:

```
MyClass:myParam
  ^(self new) MyClass:myParam
```

The instance method will execute the code as it is written in the actual Dart constructor. In Smalltalk it would look like this:

```
MyClass:myParam
  myField := myParam
```

Of course we will not write the Smalltalk code, it will be just the bytecode that behaves this way. This means that the class instance method will contain automatically generated bytecode that pushes class object on the stack,

sends the *new* message which returns the newly allocated instance of our class and then sends a message executing the actual initializer(constructor) - the instance method.

With this approach we're not breaking any of the Dart features. Having a method called *new* is not possible in Dart because *new* is one of the reserved keywords. Also having a static method with the same name as a class is not possible and results in a compilation error in Dart.

2.3.1.1 Named constructors

Dart also supports named constructors. It is basically a way how to create more different constructors because Dart can only have one constructor whose name is equal to the name of the class and method overloading(having another method with the same name but with different parameters) is not supported. There might be several reasons why this is not supported as for example, the dynamic typing. Because of this limitation, every class member has to have a unique name, otherwise there would exist an ambiguity when calling methods or functions.

An example for the named constructor might look like this:

```
MyClass.myNamedConstructor(myParam) {
  myField = myParam;
}
```

Implementation of named constructors could be the same as with the simple constructors. Creating the class instance method with the name of the constructor which has automatically generated bytecode just allocates a new instance and calls the instance method that corresponds to the actual constructor in Dart.

2.3.1.2 Redirecting constructors

Dart supports redirecting constructors. These redirecting constructors have to call another non-redirecting constructor in the same class and they're not allowed to have a method body. This might be useful in cases where we want to provide a convenience constructor that sets some default values which are used in the other constructor called by the redirecting one. The constructor call appears after a colon (:). [5]

This implementation is quite easy. If we have a redirecting constructor, there's only one call to the other constructor that has to be generated in our instance constructor method. The class instance method stays the same as in the previous cases.

2.3.1.3 Inheritance

Dart classes do not inherit constructors. If no constructor is defined in a subclass a default one is generated and a default constructor or a constructor with no parameters must exist in the superclass.

When an instance of a Dart class is being created by calling a particular constructor, a superclass constructor is called at the very beginning of the currently called constructor. There's a special syntax for specifying the super constructor that should be called which is outside of the constructor body. Constructor definition with specified named super constructor in Dart has the following code:

```
MyClass() : super.aSuperConstructor("aParam") {
  // MyClass constructor body
}
```

If the super constructor is not defined explicitly in the code a default superclass constructor is called. This behaviour has to be the same in our implementation and an automatic call to the super constructor has to be generated if no super constructor was called.

The fact that Dart constructors are not inherited is not reflected in our approach with class instance methods creating new instance. This is because in Smalltalk a class instance method can be called also from the subclass which is a different behaviour compared to Dart constructors. Therefore we have to either rely on the compiler producing an error in the compilation process or generate additional bytecode checking if the class instance method allocating new instance was called from the correct class.

2.3.1.4 More syntactic sugar

There's one more thing to care about in the constructors and that is automatic assignment of constructor argument to the instance variable. This removes some of the boilerplate code that is very common in constuctors. As an example let's consider the following code:

```
MyClass(argument1, argument2) {
    myInstVar1 = argument1;
    myInstVar2 = argument2;
}
```

This code can be written in just one line if we prepend our constructor arguments with *this.* keyword as in the following code:

```
MyClass(this.myInstVar1, this.myInstVar2);
```

Support for this syntactic sugar is not very difficult. All we have to do is to ensure the correct parsing of a constructor with this automatic instance variable assignment and that AST is correctly generated for this assignment.

Compiler will generate instance field assignment for these fields prepended with *this.* keyword and this feature becomes supported.

2.3.2 Fields, getters and setters

As already mentioned, Dart classes may have instance variables and static variables. These are very similar to Smalltalk's instance variables and class instance variables respectively. However, Dart differs in the way how these variables can be accessed. To make the behaviour the same, we will have to automatically generate accessor methods for the defined fields in the class. It means no more than generating bytecode for returning the instance or static variable from the getter method and assigning value from parameter to the instance or static variable in its setter method.

We definitely cannot omit the computed getter and setter methods. We have to parse their body which might be just a single line or a whole block of code. The body will be parsed exactly as if it was an another instance or static method. When a field is accessed, a getter(or setter respectively) is always called so the behaviour will be as it if was just another method. There is just one more thing that the parser has to take care about and that is the number of expected parameters when calling getter or the setter method. The setter method expects exactly one parameter(when assigning a value) and getter doesn't accept any parameters.

2.3.3 Control flow

Dart control flow, when compared to Smalltalk, is quite different. Smalltalk does its control flow always through message sends to other objects while passing them a block that should be executed under the certain conditions.

For example a simple conditional statement in Smalltalk is done through a call to the condition object which might be a block or a method call returning a boolean value which is either *True* or *False* object. *True* and *False* classes have each its own implementation of the *ifTrue:* and *ifFalse:* method. The implementation in the *True* class just executes the block passed in the *ifTrue:* message send. On the other hand, *ifFalse:* implementation does nothing, because the returned object was *True* object. *False* implementation behaves conversely.

The *if* statement might be easily generated with jumps. Focusing on the condition we have to generate a jump over the *if branch* that occurs if the condition wasn't fulfilled possibly landing on the *else* branch. When talking about the *else* branch, we must not forget to generate a jump over the *else* branch at the end of the *if branch.* Otherwise both branches would be executed if the condition was fulfilled.

Smalltalk/X loops are part of blocks and they are implemented natively. It has support for *whileTrue:*, *whileFalse:*, *doWhile:*, *doUntil:* and also *loop*,

which is an infinite loop. To implement support of Dart loops we have to consider its *for* loop, *while* loop and *do-while* loop. All of these loops can be implemented using jumps as well. As for the *for* loop, we have to generate a jump back to the condition at the end of the loop so it can be evaluated again. If the condition is not fulfilled we have to jump to the first statement after our *for* loop and that means generating a jump again. Apart from the condition, we have to take care about the initialization part of the *for* loop and about the statement that has to be executed at the end of every iteration just before we come to the jump at the end of the *for* loop.

While loop can be done exactly in the same way but we don't have to consider initialization part nor the statement execution at the end. There's just a jump at the beginning of the while loop which is executed if the condition wasn't fulfilled and jumps at the end of the while loop. At the end of the while loop, there's always a jump that jumps to the condition so it can be evaluated at the end of the *while* loop.

Do-while loop is even simpler and we need only one jump which is at the end of our *while* loop and jumps to the beginning of the loop if the condition was fulfilled. Otherwise the execution continues by execution the next bytecode.

Smalltalk neither supports *break*, *continue* or *switch* statements. But it's not very hard to support them with a specific bytecode sequence. As for the *break* statement, we have to generate a jump to the end of the current block. The *continue* statement has to be generated as a jump to the start of the current block e.g. a *for* or *while* loop, so the condition might be evaluated again.

2.3.4 Libraries

As already mentioned, Dart creates implicit libraries even if the library is not defined explicitly by the *library* keyword. This is actually good for our implementation because we will have to always create a class representing a library. The reason we have to create the class is that we need to store somewhere the top level variables and functions. We can treat the top level function as class instance methods and top level variables as class instance variables. Parser will then generate corresponding AST nodes when accessing top level variable or executing a top level function. Library classes will have to have a unique name within the namespace to avoid any possible conflict in the naming. If a library wasn't defined at the beginning of the Dart source file, the automatically generated library name with the Smalltalk/X namespace can be based on the current filename.

We have to take a special care about the *main* function. If the *main* function was defined in one of our source files, we have to treat this as a beginning of the execution path and if we want to execute the code after the compilation, we just want this method to be called.

2.3.5 Methods

We already know that we're going to use instance methods of Smalltalk classes as an equivalent for the instance methods of Dart classes and that static methods will be generated as equivalent class instance methods. However, Dart has more feature when defining and calling methods and we have to make this support in the Smalltalk/X.

A first difference is right in the method declaration. Method parameters in Dart might be optional whereas Smalltalk relies on the selectors that must always have the correct number of parameters. This means that the parameters in Smalltalk have to be always passed otherwise it's a different method(i.e. different selector) that is called.

Dart has also named parameters which means that a method parameters might be assigned by its name rather than its position. However, optional and named parameters cannot be present at the same time in one method declaration. Optional or named parameters must be defined after non-optional(positional) parameters.

When parsing and compiling the Dart code we don't necessarily know the type of the object that we are calling a method on. This means that we don't have all the information needed for making a correct call just by generating a message send in the compile time. The missing part is the argument that might be optional or named. For this reason we need to check in runtime what method is going to be called based on the class of our object. When we resolve the method in runtime and it has optional parameters that weren't provided in the method call, we have to pass the default values defined in the method declaration. As for the named parameters we can use just one parameter which is a dictionary containing name - value parameter pairs. These parameters then have to be loaded as method arguments correctly.

It's a bit easier when calling a static method as at this point we always know the right class and method that is being called so we can directly resolve the correct symbol and also arguments passing is easier. However, we can keep this behaviour the same as with the instance methods if the implementation would be easier.

2.3.6 Functions and Closures

Dart similarly as Smalltalk supports closures. As a pure obejct oriented language, function is also an object whose type is *Function*. Smalltalk method's class is *Method* and there is a different type for closures which is *Block*. Blocks can be passed as parameters to the other functions or stored in the variables as Dart functions can be. When implementing the support of the closures we have to specifically deal with the methods. The difference is in assigning a method object to a variable and then calling it. In Dart, we have a function object which can be called but in Smalltalk, we need to send a message which

executes the method. The solution is to actually pass a selector instead of the method object, which is then called on a current instance in case of the instance method or on the class instance object in case of a class instance method.

2.3.7 Asynchronous support

Dart has several features to support asynchronous programming. The most commonly used of these features are *async* functions and *await* expressions.[5] The *async* keyword is used to create an asynchronous method whose body will be executed in the background and a special *Future* object is returned. At the time the method returns, none of its code was executed. When the code finds an await statement, the execution is paused until this awaited object is available.

Smalltalk/X uses *Process* class to execute code in the background, so the support would mean to actually simulate async and await behaviour using this class providing a different callbacks after the execution of the async code was finished.

However, Dart asynchronous support is a broad topic and it exceeds the size of this thesis and therefore we will only run our code in a single threaded environment without asynchronous support.

2.3.8 More syntactic sugar

Dart has two operators that let developer concisely evaluate expressions that might otherwise require *if-else* statements.[5] One of them is a well known ternary operator: *condition ? expr1 : expr2*. If condition if true, *expr1* is evaluated and the result value is returned; otherwise, *expr2* is evaluated and its result value is returned. Support in Smalltalk is easy and it's the same as generating *if-else* statements.

Second operator is *??* and the whole expression has the following structure: *expr1 ?? expr2*. If *expr1* is non-null, its value is returned; otherwise, *expr2* is evaluated and its value is returned. To support this, we need to store the result of the *expr1* into a temporary variable and generate a condition checking if the stored value is null. If it is null we return this stored value; otherwise, we execute and return the value of the *expr2*.

Dart also simplifies the process of null checking. A developer doesn't have to write an *if* condition checking if the current object is not null if he wants to execute a method or access a field of this object. There's a special operator for the conditional member access: "*?.*". For example we can use it like this: *myObject?.myField = 4;* or like this: *myObject?.myFunc();* This operator checks if the leftmost operand is non-null and executes a method if there's a live object. To implement this support, we have to always store the current leftmost result into a temporary variable, generate *if* condition checking if the

value was non-null and if so, we execute the next call on the result object. There is an important thing to take care about in the implementation of the assignment expressions. We have to first pass the null checks at the left hand side of the assignment expression and only if all the null checks were passed we can safely execute the right hand side of the expression. Otherwise, if a null value was found while doing the null checks, the right hand side expression must not be executed.

There are also other operators, like *//* for direct array access, and we can simulate those by generating a particular method call. For example, for the *//* operator, we need to generate a method call that accesses the item in the array.

Chapter 3

Realisation

In the realisation chapter, an implementation of the parse and compilation process will be discussed with all the specific issues and details needed to consider while implementing the Dart support in the Smalltalk/X environment. Implementation was inspired by the parsing process used in the Dart SDK project.

3.1 Main classes

Implementation uses the following classes for parsing and compiling the Dart code:

- *DartScanner* takes care about creating tokens from the source code.
- *DartParser* parses libraries and classes and generates AST from the source code.
- *DartCompiler* is used for the actual bytecode generation from the AST provided by the *DartParser*.
- *DartParseNode* is the base class of other AST nodes created by *DartParser*.
- *DartParseClass* and *DartParseLibrary* are classes that are created by *DartParser* and they hold information about parsed classes, libraries and its methods, fields and functions.
- *DartMetaclass* is a subclass of Smalltalk's *Metaclass* and is used as a Metaclass of all the generated Dart classes.

3.1.1 Compilation process overview

Compilation process begins by creating an instance of the *DartCompiler* class. Compiler creates a *DartParser* instance which then creates an *DartScanner* instance. When creating the compiler instance, a source code path is passed to the compiler which passes it down to the parser and scanner. For reading from the source file a *FileStream* is created based on the filename and parser, scanner and compiler are initialized.

Then by calling *compile* method on compiler instance, the whole process of scanning, parsing and compiling begins. Compiler calls first parser's *parse* method which parses all the code and creates the AST. While parsing the code, parser continuously asks scanner to get a next token. Scanner reads the source code from the file stream and creates new token every time the parser asks for it until the end of file is reached.

When the parsing is finished, *compiler* loads the classes first by creating equivalent Smalltalk classes through *DartMetaclass*. It creates the superclasses first until a default *Object* class is found to ensure that new classes have always a correct inheritance hierarchy set up.

When a class is created, it continues by taking the parsed methods and generating appropriate bytecode for these parsed methods, constructors and generates also default accessor methods for getters and setters. Finally, it installs the compiled method into its owning class.

3.1.2 DartScanner

DartScanner is used to read source from the source file and to create tokens based on what it reads and returns these tokens to the parser. Scanner also saves the read tokens so they can be easily accessed when performing the second run of the parsing after the top level parsing was done.

3.1.3 DartParser

Parser performs the syntactic analysis based on the scanned tokens by *DartScanner*. It parses the top level first which means whenever it finds a function it parses only its declaration or if it finds a class it parses its definition and it adds the new function or class to the current library(which was created automatically or defined by the *library* statement). When the top level parsing is finished, it parses the definitions of function and methods and creates an AST for each function or method.

3.1.4 DartParseNode

As a base class for all the AST nodes generated by the parser, it is designed to contain information shared within all other nodes such as position in the source code. With its *accept:* method it also makes the base for the visitor

design pattern that is used in compiler for simple bytecode generation. This method is overridden by all its subclasses to call the correct visitor's method.

3.1.5 DartCompiler

Compiler, as already mentioned, uses a visitor design pattern to generate bytecode. But first thing it does is creating equivalent Smalltalk classes, its superclasses and constructing correct inheritance hierarchy. Then it generates default getters and setters for defined class' fields and finally it generates bytecode for each method in the class. Compiler also creates the library class and generates bytecode fot the top level functions.

3.1.6 DartParseClass

Instances of the *DartParseClass* along with the *DartParseLibrary* class instance are created by parser. There is always one library class representing the current library and then there might be many classes that were defined in the source code.

DartParseClass contains all the information about the class needed for its compilation like its superclass, all the instance fields, static fields, computed getters and setters and of course parsed constructors and methods as well. Parsed methods are instances of the *DartParseMethod* class. This class contains all the information about the defined method like number of arguments, whether it is static, native, starting position in the source code, used literals an so on. These classes are provided to the compiler with all the information needed for the bytecode generation.

3.1.7 DartMetaclass

DartMetaclass is there just as a base point for creating new classes compiled by the compiler. It can provide also custom compiler class to the Smalltalk/X IDE by overriding the *compilerClass* method which would allow developer to write the Dart code directly using the Smalltalk/X code editor but this feature is not currently supported.

3.2 Parser implementation

DartParser is doing a lot of important work and there's a lot of things to be clarified. We can start by the implementation of the top level parsing. Top level parsing starts by the *library*, *import* statements parsing or a top level variable / method declaration parsing or a class definition parsing. We will discuss the details of each one of these in the forthcoming sections.

3.2.1 library and import directives

Dart has a great way of creating library packages, however, there is a lot to implement including the correct organisation of the library files in a directory, supporting pubspec.yaml files and we will therefore provide just limited support which can be extended later as we don't necessarily need all the functionalities to present the Dart support in Smalltalk/X.

If a library directive is specified, we take the specified name and set a new name to the library instead of generating a unique name based on the current path of the file that is being parsed. All the parsed classes are then assigned as part of this library.

Import directives require a URI to be specified and a library is imported based on this URI. For Dart built-in libraries, the URI has a special *dart:* scheme. For other libraries a *package:* scheme is used. This scheme specifies libraries provided by a package manager such as the *pub tool*.[5] However, creating a support for the package manager is out of scope of this thesis and therefore we support only the base *dart:* scheme for built-in libraries. By default, *dart:core* library is imported automatically in Dart and so we import it automatically as well. Example *URI* may look like this: *import 'dart:io';*. Dart also support lazy loading of libraries using the *deffered* keyword, which in our case doesn't make sense as we're supporting only dart built-in libraries and we're going to parse all the code before it is executed and therefore lazy loading of the libraries is left out.

3.2.2 Top level functions declarations

When a top level function is found, we parse its declaration only as we may find in the body references to classes or other top level functions or identifiers whose declarations or definitions were not parsed yet. The same principle applies when a function(method) is found while parsing the class definition.

Parser raises an error if the top level function is marked as static which is not allowed in Dart. However we still mark the top level function as static as we want to store it as Smalltalk's class instance method which belongs to the current library class.

What exactly is parsed in the declaration is discussed in the section 3.2.4.

3.2.3 Class definitions

When a class declaration is detected, while parsing the top level, parser checks first whether this class was already referenced before(e.g. as a superclass of another class) by searching for it in the parser's *pendingClasses* dictionary. If it is found there, we take if from the dictionary; otherwise, we create new instance of *DartParseClass* and parse its definition. This means parsing its superclass first or making the base *Object* class as its superclass.

If the superclass was specified but its *DartParseClass* instance wasn't found, parser creates new *DartParseClass* instance for this superclass and adds it to its *pendingClasses* dictionary. Then the class definition is parsed after the expected curly brace "{". We are expecting here class members declarations which means a static or instance fields, computed getters or setters or functions(methods).

If a field is found and correctly parsed it is added to the *fields* dictionary of the class that is currently being parsed. If a method is found, its declaration is parsed, instance of *DartParseMethod* is saved in the *methods* dictionary and the definition will be parsed in the second run. Similarly, the declarations of computed getters, setters and constructors are parsed and instances of *DartParseMehtod* are saved in the getters, setters and constructors dictionaries respectively for later parsing of theirs definitions. After the declarations, we just skip to the matching closing brace(or skip expected semicolon in case of an abstract method) and continue in searching for next class member to parse.

3.2.4 Function declaration

In the function declaration we check whether it is static, what type it returns, parser gets the function name and then we parse its parameters. Parameter may have its type specified which we just skip as the decision in the analysis part was not to perform the type checking.

There is a special case while parsing the arguments if the current function is a constructor. We have to take care about the syntactic sugar for the automatic assignment of the constructor's parameter to an instance variable if the parameter name is prepended with *this.* keyword. Another special case related to the constructors is the initializer list. The initializer list might be specified after the parameters declaration and colon "*:*" right where a call to a super constructor might be specified as well but the super constructor call has to be the last in the initializer list. The initializer list is supposed to be used for initialization of instance variables and is particularly useful for initialization of final fields that cannot be initialized in the constructor's body. However, these calls and initializations have to be parsed later after the top level parsing is done as it also might contain calls or references to identifiers that haven't been specified at this point.

3.2.5 Functions definitions

The parser is ready to parse the function definitions after the top level parsing was finished. Scanner has read all the tokens by now and they are saved for reusing when parsing the function or method definitions. Parser therefore takes every parsed class and iterates over the parsed method declarations. Every method is checked whether it is a constructor as there's are special cases when parsing constructor first and constructor parsing is done separately.

Particular definition parsing is performed after these initial checks and every statement will be discussed in the forthcoming subsections.

3.2.6 Constructor definition

As already mentioned we have to parse the automatic field initializers first. After the parsing of parameters the parser already knows which one of the parameters is marked as an initializer. So we iterate over the parsed parameters that are initializers and generate first AST nodes. We need to load the argument and store it in our field. For this purpose we are using a *DartStoreInstanceFieldNode* which takes the field that we want to store an argument's value in. We create an expression that provides the value as well and store it in the node. This expression has to load the particular argument from the call which is represented by the *DartLoadMethodArgumentNode* where the particular argument that should be loaded is stored.

When the field initializers are parsed and required AST nodes are created, we move to the initializer list. Here we can initialize final fields as well as other instance fields using an expression, e.g. a call to some static method or just a literal value. For each of these fields we create a *DartStoreInstanceFieldNode* and we parse an expression that follows for each of these initializers.

At the end of the initializer list a super constructor call can be specified. We might call a named constructor, e.g. "super.namedConstr(myParam)", and we can parse it as a regular super call where a *DartSuperSendNode* is created. Then we just verify that the called method is a constructor. Another option is calling a regular super constructor, e.g. "*super(myParam)*", and then we have to create super send *DartSuperSendNode* where we specify the superclass' name as the method name. In both cases the required parameters are passed as *DartArgumentListNode* instance and saved in the created *DartSuperSendNode* instance. Each parameter in the *DartArgumentListNode* might be an expression.

After we've dealt with this special statements the rest of the constructor's body is parsed as a regular method with a sequence of statements.

One more thing to care about is the redirecting constructor and there we have to generate just a regular method call to the constructor represented by the *DartInstanceCallNode* that will contain the arguments list.

3.2.7 Identifiers resolving

One of the first things that the parser has to deal with is resolving of identifiers. Inside functions we create different scopes based on the enclosing blocks where a defined local variable or loaded method arguments can be found. Parser tries to resolve an identifier in these scopes first by taking the current scope first and if the identifier wasn't found, it continues to search in the parent scope representing a parent enclosing block. If the parent scope is null, we are in

the first top function block and there is no other parent scope. If the identifier was resolved in the scope search we have to generate an AST loading either a method argument represented by *DartLoadMethodArgument* class or local variable represented by *DartLoadLocalNode*.

If the identifier still wasn't resolved, parser tries to resolve the identifier as an instance field or method of the current class. If a field was found, AST node for calling a getter is generated either as a *DartStaticGetterNode* if the field is static or a *DartInstanceGetterNode*. If the resolved identifier was a method a *DartPrimaryNode* is returned with the resolved method as its primary object. This primary node is then used later when parsing selectors(i.e. when parsing a possible sequence of calls like *method1().method2();*) and it is converted to the actual method call represented by *DartInstanceCallNode*.

Otherwise parser has to resolve the identifier in the global or top level scope. This might be a class or a variable or a top level function and therefore it is again returned as a *DartPrimaryNode* with the resolved identifier and the primary node is again processed later when parsing selectors.

However, if the identifier wasn't resolved even after searching in the global scope, we search for an identifier in the imported libraries and return a *DartPrimaryNode* instance again. Otherwise, we raise an error because the identifier wasn't found and therefore it could not be resolved.

3.2.8 Conditions AST

If we have found the *if* keyword we have actually found a beginning of a condition and we expect an expression enclosed by parentheses to be present. Therefore the condition is parsed and its AST is saved in a temporary variable. Then if a left brace follows we parse a sequence of statements; otherwise, we parse just a one statement which is the true branch that will be executed if the condition was fulfilled. We store the saved AST in a temporary variable again and then we parse the else branch if the *else* keyword is following right after the end of the true branch. Finally, we create an AST node for the condition which is represented by the *DartIfNode* that contains the AST of the parsed condition, AST of the true branch and possibly the AST of the false branch.

3.2.9 Loops AST

In the loops section we have to distinguish parsing between the *for* and *while* loops and also *do-while* loop. We parse all these statements separately.

3.2.9.1 For loop

For loop starts with the *for* keyword and after the keyword, we expect a left parenthesis to be present. After the parenthesis, we check whether a variable declaration follows and if so, we first parse a list of the variable declarations.

If the variable declarations does not follow, we try to parse an initializer expression instead.

After the initializer, we expect a semicolon and the actual condition that is evaluated at the beginning of every iteration and if it is not fulfilled we jump to the next statement after the loop. When the condition is parsed, we try to parse the *increment* expressions that are executed after every iteration and then the closing parenthesis is expected to be present.

If we have all the ASTs of the initializers, condition and increment expressions parsed correctly and stored in the temporary variables we move to the parsing of the *for* body which might be just one expression or a sequence of expression if a left brace is following.

If the body parsing was successful, we create the final AST representation of the *for* loop which is done by creating an instance of *DartForNode* that contains the AST of initializer(either variable declarations or initializer expressions), the *for* condition and the increment expressions.

3.2.9.2 While loop

While loop starts with the *while* keyword followed by an expression enclosed in parentheses. Therefore we parse this expression and save its AST in a temporary variable. Then similarly to *for* loop parsing, we parse the while loop body which might be one statement or a sequence of statements if the left brace "{" follows. At this point we have successfully parsed the the while statement and we create an AST represented by the instance of *DartWhileNode* which holds both the condition and the while loop body.

3.2.9.3 Do-While loop

Do-while loop parsing is very similar to the while loop parsing, but instead of the *while* keyword, we have to find the *do* keyword first followed by a left brace "{" with the body. We keep the parsed body in a temporary variable which is a sequence of statements. After the body a *while* keyword is expected followed by the condition in enclosing parentheses. Parsed body and condition is then used for initialization of a *DartDoWhileNode* instance which represents our currently parsed *do-while* loop.

3.2.10 Function calls

There are several scenarios of different function calls. It might be a top level function call, call to an instance or a static function of the current class and also an instance call of the function(method) of another class or a static call of another class' static function(method).

If we're calling a top level function, we have resolved a *DartParseMethod* instance for this function and as we are treating top level functions as a static methods of its owning library, we generate a *DartStaticCallNode* with this

resolved *DartParseMethod* instance and its owning library as a receiver of this method call.

In case of a current class' function, either static or an instance, we have direct access to its *DartParseMethod* also. Therefore we generate either *DartInstanceCallNode* with a special *this* receiver which represents the current instance or a *DartStaticCallNode* with the receiver set as the current class instance.

However if we are not calling an instance function(method) of the current class, the receiver has to come from some variable or another previous function or method call. As we are not parsing the types, we have to generate a generic message send based on the selector. The selector is created out of the function's name and arguments that are in the current call. We represent it again by *DartInstanceCallNode* whose receiver is an AST node coming from the parsed preceding expression. The method name that should be called is stored inside the node instead of the *DartParseMethod* instance.

In case of a static call to a function(method) of another class, a name of the class has to be presented before the actual method call and therefore we are able to resolve directly the instance of *DartParseMethod* which contains information about the method needed for generating a *DartStaticCallNode* and the receiver is the resolved class.

For all these calls we have to parse arguments as well. They are parsed as expressions and there is an AST stored for each argument. These parsed arguments are stored in the *DartInstanceCallNode* or *DartStaticCallNode* instance.

There is one more thing to care about and those are the super calls. If a *super* keyword is found we know that we are sending a super message. Method is resolved by its name in the superclass and arguments are parsed in the same way as for the regular instance or static calls. Super call is then represented by an instance of *DartSuperSendNode* with stored resolved method and parsed arguments.

3.2.11 Literal objects

Literal objects like numbers or strings are recognized by *DartScanner* first. It creates a token with the literal value which is always a string but the token's type is set to *#kDouble*, *#kInteger* or *#kString* based on what literal type was read.

Parser processes these values further based on the token's type and it creates a particular object out of the token's string value. For example, it creates a *DartInteger* instance if the token's type is *#kInteger* and initializes it to a value created from the token's string value read by the scanner, e.g. "123". After instantiating and initiliazing a correct object, parser creates an instance of *DartLiteralNode* with the correct literal value. This value is still

the parser's object and is not a native implementation of the Double, Integer or String. It is converted later into a native object by the compiler.

3.2.12 Constructor call

Constructor call starts with the *new* keyword followed by the constructor's name, which might be just a class' name or a class name followed by a dot and a name of the named constructor. Parser also takes care of skipping possible definition of a generic type. After the constructor is resolved, arguments are parsed exactly as other function arguments. Resolved class, constructor name and arguments are stored in the *DartConstructorCallNode* which represents a constructor call.

3.2.13 Default constructors

If no constructor was specified in the class, parser generates a default constructor method whose name equals to the class name and generates a default AST sequence for the constructor. This means that we create an instance of *DartParseMethod*, we set its *owningClass* field to the class that we are generating the constructor for, arguments are set to nil value as default constructor takes no arguments. The name of the method is the same as the class name. The body of the generated constructor is a sequence node containing just a super call to the resolved super constructor with no arguments which has to exist. Otherwise, an error is raised because a non-default super constructor has to be called.
Generated default constructor as a *DartParseMethod* instance is finally added to the class' constructors dictionary.

3.2.14 Getters and setters

Getters and setters that are generated have a very simple body. Every field that was parsed is checked whether it has a getter and setter and if they are missing, we generate their AST. The AST for getters consists of a single *DartReturnNode* which returns a field value loaded on the stack either by *DartLoadInstanceFieldNode* for instance fields or by *DartLoadStaticFieldNode* for static fields.

3.2.15 Native classes

To inform parser what native classes are available in the runtime, we create equivalent instances of *DartParseClass* and we fill them with *DartParseMethod* instances containing just the declaration of the methods. This class has already set its Smalltalk class so the compiler doesn't overwrite our class and the methods are marked as native, so they won't be compiled either. By default, the *Core* library is always imported and so it is imported even

without specifying the *import* statement. With this configuration, parser will not complain about non-existent classes and it will create the correct AST just as we need.

3.3 Compiler implementation

DartCompiler is responsible for creating new equivalent Smalltalk classes and for doing the final compilation of the method's AST returned by the parser into the Smalltalk/X bytecode. Compiler goes over all the classes that parser saved in the top level scope. It compiles each class but it skips the native ones and finally compiles also the library which contains all the top level functions.

Compiler uses the *Visitor* design pattern to walk through the generated AST nodes. Our compiler takes care of generating the symbolic code which is an intermediate representation of the code that is more human readable than the bytecode. Final compilation from the symbolic code to the bytecode was copied from the Smalltalk/X compiler which uses this technique.

3.3.1 Class creation

As already mentioned compiler creates classes and its superclasses first to ensure a correct inheritance hierarchy. So if a Smalltalk equivalent of the class wasn't created yet, compiler creates a new class by calling a method *createDartClass:* of *DartMetaclass*. After this step it loops through all the methods, getters and setters and compiles them. If the method is a constructor, compiler generates a special static method to create a static part of the constructor first and then compiles the constructor method.

3.3.2 Constructors compilation

We have already mentioned that compiler is generating a static part of a constructor. This part is very simple and we just create a *DartAllocCallNode* instance and we pass it the current constructor method so when generating bytecode we know what non-static constructor has to be called. This instance of *DartAllocCallNode* is wrapped in a *DartReturnNode* so the bytecode for returning the newly initialized instance is called.

As for the actual bytecode that is generated while visiting the *DartAllocCallNode* instance, we first store a literal symbol - *#new* message and we generate the following sequence of symbolic code to create a new instance of the current class:

```
#pushSelf
#send0
linenumber (e.g. 42)
literal index (#new message)
```

So we first push a self object on the stack which is a *class class instance* and then we generate a message send to this class class instance. The *send0* symbolic code means that we are sending a message with 0 arguments to the object that is on the stack. The *send0* code is then followed by the line number in the original source code whose value is not important in our case as this is a generated piece of source code. Last symbolic code is the index of a literal stored in our method which is the *#new* message.

When we have a new instance on the top of the stack we call the correct parsed non-static constructor. We first push all the arguments on the stack that were passed to this static part of the constructor with the symbolic code *#pushMethodArg* which is followed by the argument index. When the arguments were pushed on the stack we generate the actual message send to the correct constructor which is done through the *send* symbolic code again followed by a line number in the original source code and the index of the literal which is a selector symbol for the current constructor, e.g. #MyClass if the constructor is not a named one.

Finally, we have an initalized new instance of our class on the top of the stack and as this *DartAllocCallNode* is always wrapped in a *DartReturnNode*, symbolic code *#retTop* for returning top of the stack is added at the end of our symbolic code.

After this we just generate the bytecode out of the symbolic code by calling the *generateByteCode:* method which has exactly the same implementation as regular Smalltalk/X compiler.

3.3.3 Constructor calls

As described in the section 3.2.12, *DartConstructorCallNode* is created when parsing a call to a constructor. For the actual symbolic code generation, we call the static constructor method which allocates a new instance and calls the correct instance constructor(initializer) method, which results in the following sequence:

```
#pushGlobalS
literal index of the class
arguments load
#send
line number
literal index of the selector symbol
number of arguments
```

So we have to save the current class name as a symbol in the method's literals first to reference it after the *#pushGlobalS* symbolic code. This name is constructed as a namespace plus the class name, e.g. *#MyLibrary::MyClass*. When we've pushed this symbol on the stack we're ready to generate a message *#send*. This again consist of loading the arguments that should be passed

to the method, the *send* symbol, line number in the original code, index of the selector symbol in the method's literals array and a number of passed arguments.

If the *DartConstructorCallNode* doesn't have any parent AST node the value is then dropped from the stack right after the new instance was created. Otherwise the new instance was pushed on the stack and will be used by some other statement.

3.3.4 Method calls

We have already introduced method calls in the constructor calls. However, we will go a bit deeper discussing different versions of the *#send* symbolic code in this subsection and we will differentiate between the static and instance calls.

As for the static calls, we have to do exactly the same steps as when we were calling the static part of the constructor. So we have to create a class name as a symbol and we push it into the current method's literals array first, add *#pushGlobalS* symbolic code followed by the literal array. This will be the receiver of our message send - a class class instance as it's a static call. Then we push the arguments, e.g. a literal, variable, or a method call, depending on the argument's AST. After the arguments we add the *#send* symbolic code or one of its variants.

When making an instance call, we have to visit the receiver node first which might be just a loading a local variable but also making a lot of subsequent calls to get the actual receiver loaded on the stack. The receiver's AST is stored in the *DartInstanceCallNode*.

After the receiver is loaded which is either a class symbol for the static call or another receiver for the instance call, we can proceed to the actual message send generation. Smalltalk has a few different variants of the message send symbolic code. The general one, *#send*, is followed by a line number, a literal index of the method selector that should be called and number of the arguments. If there are not many arguments to be passed in the call, we can generate a specialized version of the *#send* symbolic code which is one of the *#send0*, *#send1*, *#send2* or *#send3*. All these codes are meant to save one bytecode and improve the performance as we know directly from the symbolic code / bytecode how many arguments were loaded before we called a method.

Another specialized message send version is a *#sendDrop0* and its siblings with suffixes *1*, *2* and *3*. This specialized message send also drops the returned value from the top of the stack without generating an explicit code to drop the value. Dropping the returned value is needed if the result of the call is not used or if the returned value is void.

There is also a shortcut for sending a message to the *self(this)* object. We don't have to explicitly push a *self* object on the stack, but we can use the *#sendSelf* symbolic code. It just simply sends the message to the current object whose method is being executed. It has the same following symbolic

codes as a regular send and there are also specialized variants with defined arguments count, e.g. *#sendSelf0*. As for dropping the value for void or unused returned values from method calls, there are versions for dropping the value from the top of the stack exactly as in regular *send*. Those are *#sendSelfDrop0*, and variants with suffixes *1*, *2* and *3*.

An example of an instance method call with name "*myMethod:*" whose receiver is stored in a local variable and passing a method argument as a parameter will look like this:

```
#pushMethodVar1 (receiver in local variable)
#pushMethodVar2 (method argument)
#send1
line number
index of the method symbol literal (#myMethod:)
```

Assuming that our receiver is stored in a local variable of our method at index 1, we push it on the stack by generating the *#pushMethodVar1* symbolic code. Then we have to generate the code for loading a method argument, which, as we defined, will be stored in another local variable, e.g. at index 2. We do this again by generating *#pushMethodVar2* symbolic code. After this we generate the actual message send. We know that we have only one argument, so we can use the *send1* symbolic code to call the method. We have to add the line number and finally we add index of our method symbol *#myMethod:* in the method literals array.

An example of static call with name *myStaticMethod:* that takes one argument and whose receiver is a class called *MyClass* might look like this:

```
#pushGlobalS
literal index of the class
#pushMethodVar1 (method argument)
#send1
line number
index of the method symbol literal (#myStaticMethod:)
```

Exactly the same call was already introduced while describing the call to a static constructor part in the section 3.3.3. We push the receiver which is a global symbol of our class. Then we push the argument(assuming that it was stored as a local variable) and we generate the message send exactly as in the previous instance call example. The only difference is in the receiver loading part.

3.3.5 Getters and setters

Automatically generated getters and setters have a generated AST as described in the section 3.2.14.
As for the instance getters, we have all the information needed for compilation stored in the DartLoadInstanceFieldNode which is just a reference to the

DartClassField instance that should be pushed on the stack. Let's assume that the instance field is the first one in the class. The compilation then results in the following symbolic code: *#pushInstVar1*. Just one code is enough. However, if the field index is greater then 10, there is no specialized version for pushing the field with index greater than 10 on the stack and therefore we have to use the general *#pushInstVar* symbolic code, which is followed by the index of the instance field.

The static field access is generated exactly the same way as the instance field access. The only difference is that the getter is installed into the class class object so it is accessing the class instance variable that we have decided to use as an equivalent of the Dart static field.

The following code illustrates a static field access in Dart:

```
MyClass.myStaticField;
```

This code is represented by the *DartStaticGetterNode* which contains a DartParseClass containing the getter and also the getter reference that should be called, e.g. *MyClass* and *myStaticField* getter as *DartParseMethod*. The *myStaticField* might also be a computed getter in which case it is still represented by *DartStaticGetterNode* and the particular method call for this getter is generated.

In case of instance calls, the following code might represent the instance creation and getter call in Dart:

```
var myInstance = new MyClass();
myInstance.myInstanceField;
```

This getter call is represented by the *DartInstanceGetterNode* and as we don't know what is the class of myInstance at this point, we just generate a simple message send.

A problematic piece of code is a code where a getter that returns a closure is called and the closure is executed right away. This can be done by the following code:

```
var myInstance = new MyClass();
myInstance.myClosure();
```

With this code we're not able to differentiate between the getter call with a closure execution or just a simple method call with no arguments. We have to check in the runtime whether we have just called a getter and a closure is currently on top of the stack and if it is the case, execute the closure; otherwise, don't do anything more because we have just called a simple method. This behaviour is not currently supported in the implementation of the Dart in Smalltalk/X.

Automatically generated instance setters with *DartStoreInstanceFieldNode* are processed by compiler generating first the symbolic code for the value that should be stored and right after that the actual symbolic code for storing it in

the correct field. This is done with the *#storeInstVar* symbolic code that is followed by the index of the instance field. There are also specialised variants with suffixes from 1 to 10 exactly as variants of *#pushInstVar*.

The storing of static fields is done again exactly the same way as instance fields storing. So the static setter methods have the same symbolic code / bytecode for storing the fields, which is using the *#storeInstVar* but the methods are installed into class class object so the static setters are storing the class instance variables.

3.3.6 Loops

Loops in Smalltalk are implemented through the execution of blocks. However, I've decided to generate loops with jumps as this approach is easier to implement.

3.3.6.1 For loop

DartForNode as an AST representation contains a four essential AST parts: Initialization, condition, increment and the body. If the initializer contains a loop variable, it is reserved in the current function or method as a local variable. The symbolic code of the initializer is generated from its AST and after this symbolic code we save the current position in a temporary variable where the condition code will begin as this is the place where we have to jump at the end of our for loop.
Compiler continues with the symbolic code generating for the condition. This includes also a generation of the jump if the condition wasn't fulfilled. Therefore it generates an *#falseJump* symbolic code and it reserves a place for the jump location. This location will become known once we have generated the symbolic code for the body and increment statement.
After the reserved place for the jump location we generate the symbolic code for the body and right after the body, compiler generates symbolic code for the increment statement. This statement is finally followed by the jump on the condition so it can be evaluated and checked whether we should continue iterating or stop looping.
At this point we know the position in the symbolic code where we want to jump if the condition wasn't fulfilled and we can fill the reserved place for the *#falseJump* location.

3.3.6.2 While loop

While loop symbolic code generation is simpler compared to the for loop as it doesn't contain initialization or increment but it is still very similar to the for loop. We generate the symbolic code for the condition followed by the *#falseJump* symbolic code and a reserved place for the jump location if the condition wasn't fulfilled. Then the compiler continues by generating code

for the body and appending jump to the while condition location so it can be evaluated again. Finally it fills the reserved place of the condition's jump which is the first location after the body and jump to while loop condition.

Let's consider the following code as an example:

```
while (true)
{
    // body code
}
```

This will result in the following symbolic code:

```
#pushLit1
#falseJump
location
... body code ...
#jump
condition location
```

3.3.6.3 Do-While loop

Do-While loop is the simplest one to generate. It contains the same information as the WhileLoopNode but it needs only a single jump. Compiler starts by generating the symbolic code for the body which is followed by the symbolic code for the condition. After the condition a single *#trueJump* to the beginning of the body is generated. This means that at the end of the loop the condition is evaluated and we jump to the beginning of the body if it was fulfilled; otherwise, no jump is performed and the while loop is exited just by continuing in the execution of the next bytecode instruction.

3.3.7 Conditions

When generating an *if-else* conditional statement, compiler starts by generating the bytecode for the condition. After the symbolic code for the condition is generated, it generates a *#falseJump* and reserves one place for the jump location. In case the *DartIfNode* contains also the *else* branch(the else AST is non-null), jump location has to point at the first symbolic code of the else branch; otherwise, it has to point at the first symbolic code after the generated code for the *if-body*. It is important to state that these locations are different.

Compiler therefore generates the symbolic code for the *if body* and if the *else* branch is non-null, it has to generate one more *jump* and reserve a place for the jump location pointing at the first instruction after the *else* branch which makes the difference in the locations. If the jump code was missing, the *else* branch would be executed no matter what the *if-condition* value was. If there was no *else* branch, compiler just updates the reserved location for the condition jump with the correct value as just described.

If the *else* branch is non-null, compiler continues by generating the symbolic code for the *else body* and it updates the jump location generated after the *if* body afterwards.

As an examples we can consider the following code with hardcoded condition values set as true:

```
if (true) {
    // if body
}

if (false) {
    // if body
} else {
    // else body
}
```

This results in the following symbolic code:

```
#pushLit1
#falseJump
location1
... if body ...

#pushLit2
#falseJump
location2
... if body ...
#jump
location3
... else body ...
```

In the first condition that contains only *if branch*, the hardcoded *true* literal as a condition is pushed first and the *#falseJump* is generated with location1. The location1 has to point on the first code of the second condition which is the location of *#pushLit2* and then the *if-body* symbolic code follows.

The symbolic code of the second condition starts exactly as the first one but the difference comes after the *if-body* was generated. Here we can see the *#jump* symbolic code with the *location3*. The location3 is pointing at the first symbolic code after the *else body*. The *location2* has to point at the symbolic code after the *location3* which is the start of the *else* branch.

3.3.8 Super call

Smalltalk/X uses for super calls the *#superSend* symbolic code. It requires the same arguments as the standard *message send* with one additional argument which is the super class object that owns the method object. So at first we push

the self object as an instance which we want to call the super method on by *#pushSelf* symbolic code, then we generate symbolic code for the arguments and finally the *#superSend* symbolic code is generated followed by the line number, method literal index as a symbol, number of method arguments, and the literal index of the superclass object.

Let's consider two classes A and B where B is a subclass of A and overrides a function called myFunction(arg1):

```
class B extends A {
    myFunction(arg1) {
        super.myFunction(arg1);
    }
}
```

This will results in the following symbolic code:

```
#pushSelf
#pushMethodArg1
#superSend
line number
1 (method literal index)
1 (arguments count)
2 (super class literal index)
```

Here we first push the self object on the stack which is an instance of the class B. Then a method argument *arg1* is pushed and finally followed by the *#superSend* symbolic code, the line number, method literal index(a super implementation), arguments count and superclass literal index which is the class *A*.

3.3.9 Return statement

Return statement represented by *DartReturnNode* contains only one expression that should be returned. It may be null in case of return in a *void* function. Therefore, the compiler tries to generate symbolic code for the expression whose result should be returned. If the expression is null (return in a *void* function), compiler generates *#retNil* symbolic code; otherwise, it generates a *#retTop* symbolic code so the value from the return expression is returned.

Let's consider the following code with non-void function and a simple return of a literal value:

```
int myReturnFunc() {
    return 1;
}
```

This will be compiled into following symbolic code:

```
#pushLit1
#retTop
```

The *DartReturnNode* will contain here only simple expression for loading the literal value with number 1. When the value is on the top of the stack, the function returns this value after the *#retTop* symbolic code.

3.3.10 Literal values

We've been talking about storing literals in the method's literals array but we haven't explained enough what these literal values are and how they are stored in Smalltalk/X.
Literal value can be any value that is known in compile time, e.g. a class class object, number or string. This value has to be stored somewhere, where it is available in the runtime as well. We store these values in the previously mentioned method's literal array. Then we often refer to these values in the symbolic code by their indices depending on the symbolic code / bytecode that needs the value stored as the literal.
Literals are represented by *DartLiteralNode* and there are 4 different variants of the *#pushLit* symbolic code / bytecode. The first one is *#pushLit* with suffixes from 1 to 8 for pushing literals with indices in range from 1 to 8. If the index is larger than 8 but the index still fits in one byte(i.e. value is less than or equal to 255) we have to use the *#pushLitS* symbolic code followed by the actual index. If the index value doesn't fit in one byte there's *#pushLitL* that expects number that fits into two bytes. If even this value is not enough, the last variant is *#pushLitVL* that expects a number which fits into four bytes.

3.3.11 Loading local variables

Local variables are represented by *DartLoadLocalNode* containing the variable that knows its index in the method's local variables. Compiler then generates symbolic code to push local variable on the stack which is the *#pushMethodVar*. There are also optimised variants with suffixes in range from 1 to 6 for pushing variables at these indices. If the index is greater than 6 *#pushMethodVar* symbolic code with index of the variable followed has to be used.

There is a special case when loading the self object on the stack which is an equivalent of the Dart's *this* object. If the parser finds *this* keyword it creates the *DartLoadLocalNode* with special flag indicating that the receiver is a self object. Afterwards the compiler has to handle this case. For example, when generating a method call, compiler checks whether the receiver node is a self object and if so, it uses the *sendSelf* symbolic code. If it is not a method call it has to push the self object on the stack which is done through the *#pushSelf* symbolic code.

3.4 Completeness

Completeness of the solution will be presented on the set of examples using Dart Standard Library modules. These modules were implemented natively in Smalltalk and the developer can use them in his or her code. However, not 100% of the functionality is supported for each module.

3.4.1 Dart core

Dart core library contains built-in types, collections, and other core functionality for every Dart program.

Some classes in this library, such as String and num, support Dart's built-in data types. Other classes, such as List and Map, provide data structures for managing collections of objects. And still other classes represent commonly used types of data such as URIs, dates and times, and errors.[6]

The collection library is part of the dart:core library but we'll provide examples in a separate section. We will show an examples of base core functions like *print*, *identical* and base classes like *Object* and *String*.

3.4.1.1 print

The *print* function is implemented natively as part of the *core* library. In Dart, this function prints the string representation of the passed argument object on the standard output. I have slightly modified this behaviour to match the Smalltalk behaviour and we print this object using the Smalltalk's *Transcript* class. If this was implemented to print the value on the standard output developer wouldn't be able to see the printed values unless the Smalltalk/X is run from the command line.

The implementation of this function is the following code:

```
print:anObject
  (aString isKindOf:DartCore::Object)
    ifTrue:[Transcript showCR:aString toString stString.]
    ifFalse:[Transcript showCR:aString printString.].
```

There is one more thing to mention about the *stString* method. As the Transcript needs an instance of Smalltalk's *String* class, we have to always convert this value into a Smalltalk/X String object and therefore there is the stString that converts the *core:String* object into the Smalltalk String object.

In most of the cases it is the Dart's *core:String* object that returns the Smalltalk's representation but there is an exception with the boolean objects. That is given by the limitation when we have to use the Smalltalk's *True* and *False* objects while generating *#trueJump* and *#falseJump* bytecodes. These bytcodes require Smalltalk's boolean object on the stack to get the correct jump behaviour provided by the VM. We can get an instance of standard Smalltalk class like *True* and *False* into our print function and therefore we

have to check if our object inherits from our native *DartCore::Object* class and call the right method to get the Smalltalk's string.

We can consider the following *Hello World* code as an example:

```
main() {
  print("Hello World!");
}
```

Here the compiler will create a literal for the "*Hello World!*" string which is an instance of *core:String* class. When calling the *print* method, it will get this *core:String* instance as an argument and the Smalltalk's *String* will be extracted using the *stString* call and passed to the *Transcript*'s *showCR:* method which will finally print the "*Hello World!*" string.

3.4.2 identical

The *identical* function is the Dart's way of comparing the object's identity, i.e. checking whether two instances reference the same object. In the past, Dart also had the operator === but it was removed and replaced by the *identical* function.

The native Smalltalk's implementation of the identical function is very simple using the == message send which performs the object identity check:

```
identical:first _:second
    ^(first == second)
```

Let's consider the following Dart code as an example:

```
class MyClass {}

main() {
  var object1 = new MyClass();
  var object2 = new MyClass();
  var object3 = object1;

  print(identical(object1, object2));
  print(identical(object1, object3));
}
```

We have a single class called *MyClass* and we create two instances of it stored in the *object1* and *object2*. Then we assign the *object1* to variable *object3* to reference the same instance. With the usage of the previous *print* implementation we can print out the boolean value returned by the *identical* function. The printed result is:

```
false
true
```

In the first comparison we are comparing the *object1* with *object2* which both contain a different instance of the *MyClass* and therefore the *false* value is returned and printed.
In the second case we compare the *object1* with *object3* and they both hold reference to the same object and therefore the *identical* function returns *true* value.

3.4.2.1 Object

The base Dart class, *Object*, is a part of the *core* library. In our Smalltalk's implementation it contains only the base *toString* method which might be overridden by other classes and the implementation has to return the *core::String* instance. The *print* function uses this *toString* method for printing objects. The default implementation in Object prints the current class which corresponds with the behaviour in Dart's implementation. The Smalltalk's implementation has the following code:

```
toString
  ^ String withSTString:('Instance of ''',
                         (self className),
                         '''').
```

So if we consider the following example code:

```
class MyClass {}
main() {
  var object = new MyClass();
  print(object);
}
```

We'll get the following string printed in the transcript window as a result of the execution: *Instance of 'DartmySrc6::MyClass'.*

3.4.2.2 String

Native implementation of *core::String* uses the Smalltalk's String class inside. Smalltalk's version is stored as a field called *stString*. Operations like concatenation using the operator *+* is implemented here as well as getters like length.

The concatenation of two strings has the following code:

```
+ anOtherString
  ^DartCore::String
     withSTString:(stString, anOtherString stString)
```

So it concatenates the Smalltalk's representation of String and creates a new instance of core::String with the newly concatenated Smalltalk's string.

Let's consider the following example code:

```
main() {
  var string1 = "String 1";
  var string2 = string1 + " String 2";
  print(string1);
  print(string2);
  print(string1.length);
}
```

This will result in a following output:

```
String 1
String 1 String 2
8
```

We can see that the original string in the *string1* variable wasn't modified and a new instance of the concatenated *string1* with literal " *String 2*" was created. At the end, the length of the *string1* is printed.

3.4.3 Collection

The open source Dart implementation contains interface definitions of Iterable, List, Set and Map data structures in the *core* library. However, there is no implementation provided directly in the *core* library. Base implementation is done with the help of mixins in the *collection* library. For example, the base class that can be used for implementing the *List* interface is the *ListBase* class that uses a *ListMixin* and both of these classes are part of the *collection* library.

In the core library, Dart uses factory method which has an external source code for creating instances of the *List*, *Map* or *Set*. This *external* keyword is used for a specific behaviour of the Dart compiler so it can provide different implementation(e.g. when the code is cross compiled to javascript) and *List*, *Map* or *Set* classes can be used right from the *core* library.

We don't need to provide different implementations of the *List*, *Set* or *Map*. Therefore, the factory methods are implemented directly in the *core* library which creates an instance of the class from the *collection* library which is implementing the correct interface, e.g. the *List* interface.

As an example, we can consider the following code using *List* and its operations for adding, removing and selecting elements:

```
main() {
  var list = new List();
  print(list.length);
  list.add(1);
  list.add(2);
  list.add(3);
  print(list.elementAt(1));
```

```
    list.add(4);
    print(list.last);
    list.remove(2);
    print(list.elementAt(1));
}
```

We create a new instance of List first which will return an instance of our base implementation which is backed up by the Smalltalk's *OrderedCollection* object. We print the length right after creating the list which returns an integer with value 0. Then we add three numbers, print second element in the array(at index 1), we add one more item, print last element, remove element with value 2(which is at index 1) and we verify that the list was shifted and we have number 3 at our second position now.

The output of the execution follows:

```
0
2
4
3
```

In the implementation we often just forward the calls with values to the Smalltalk's *OrderedCollection* implementation. However, we have to take care about passing / returning the right object, e.g. Smalltalk Integer instead of Dart's int object and vice versa. We also have to be aware of the different start index in Smalltalk. Indexing in Smalltalk starts from 1 whereas indexing in Dart starts from 0. We can see all this on the following implementation of the *emelentAt* and *length* methods:

```
elementAt:anIndex
    ^stArray at:anIndex stInteger + 1


length
    ^DartCore::int withSTInteger:stArray size
```

In the *elementAt* method, we get a *core::int* in the parameter but we have to pass a Smalltalk *Integer* object to the *Array* instance stored in the *stArray* field. We also increment the index to match the correct one in the Smalltalk.

In the *length* method, the value returned from the *size* call to the *stArray* is converted into *core::int* object as that's what is expected from this method.

3.4.4 io

The *io* library provides API for reading, writing, creating or deleting files, directories or links in the native file system as well as network APIs allowing developers to use web socket protocol, TCP protocol as well as writing to standard input and reading from standard output.[7]

Many of the APIs are designed to be asynchronous. However, as we are not supporting the async execution in this version of our implementation, only the synchronous versions are implemented. We can consider the following code for creating, writing a string into a file and reading it again:

```
import 'dart:io';

main() {
  var name = '/Users/branislavhavrila/dart_file.txt';
  var file = new File(name);
  file.writeAsStringSync('New content');
  var fileContent = file.readAsStringSync();
  print(fileContent);
}
```

This code creates a new file called "*dart_file.txt*" and writes one line into this file: "*New content*". Then we read the string from the file and print it using the *print* function.

The *File* class implementation in Smalltalk is using internally Smalltalk's *FileStream* for writing. The *writeAsStringSync* implementation has the following code:

```
writeAsStringSync: aString
    |fileStream|

    fileStream := path stString asFilename writeStream.
    fileStream nextPutAllUnicode: aString stString.
    fileStream close.
    ^nil
```

We get the write stream first, which truncates the file if it already existed. This matches the behaviour in the native Dart implementation. Then we write the whole string to the stream and we close it to make sure that everything gets written to the file. We have to make sure that we pass the Smalltalk's String to the *nextPutAllUnicode:* method and therefore we call the *stString* method first. Finally, we return nil as the *writeAsStringSync* has a void return type.

The *readAsStringSync* implementation has the following code:

```
readAsStringSync
  |textContent|

  textContent := path stString
                      asFilename
                      contentsAsString.
  ^DartCore::String withSTString: textContent.
```

Here we are using a very simple implementation for reading the whole content of a file directly with the *contentsAsString* method. Then we create the Dart's string instance that contains the returned string.

3.4.5 math

The *math* library contains mathematical constants like π, base of natural logarithms e, mathematical functions like *min*, *max*, *sin* and also a random number generator.

These mathematical functions are the top level library functions so they are implemented as static functions in the library class called math. Mathematical constants like π are implemented as method getters which is the usual way of implementing constants in Smalltalk.

The implementation of the π constant has the following code:

```
PI
    ^DartCore::double withSTFloat:3.1415926535897932.
```

The implementation of the functions like *max* has a very simple code as well:

```
max:a _:b
    a > b ifTrue:[^a] ifFalse:[^b].
```

However, the Dart implementation has slightly complicated implementation as it deals with the *NaN* and negative zero values. This example code uses max, min functions and prints out the PI value:

```
import 'dart:math';

main() {
  var maximum = max(1, 2);
  var minimum = min(1, 2);
  print(maximum);
  print(minimum);
  print(PI);
}
```

This simply prints out the following lines:

```
2
1
3.14159265358979
```

3.4.6 html

HTML elements and other resources for web-based applications that need to interact with the browser and the DOM (Document Object Model). This

library includes DOM element types, CSS styling, local storage, media, speech, events and more. [8]

The *html* library is operating on top of the HTML document and the source that is using the library should be included as a Dart script in the *script* HTML tag and it should have a main function. The HTML document is represented by the *HtmlDocument* class and it is automatically included as a property in the *html* library when the Dart script is executed.

The main function contains the logic e.g. for manipulating the DOM model. For example, a developer can subscribe to the click events on a particular HTML element which he can find by using the *querySelector* or *querySelectorAll* functions and then append, remove or change the content.

As this functionality depends on a web browser support, we provide only basic examples of the Element as it is out of scope of this thesis to provide all the functionality and integration of Smalltalk/X with a browser and to implement all the APIs that are included in this library. Also the Dart doesn't include the *html* library in the stand-alone VM as it is meant to be used for work with the DOM, CSS etc. in a browser. There are also many javascript objects wrapped into their Dart equivalent to provide all the functionality. Therefore, if a developer tries to include the *html* library and run the code from the command line, the following error message is shown: The built-in library 'dart:html' is not available on the stand-alone VM.

However, to show at least some basic usage of manually creating HTML elements, a developer can use the *Element* class which has many factory methods for creating them. The following code is creating a new anchor element, break line *br* and *div*:

```
main() {
  var anchor = Element.a();
  var br = Element.br();
  var div = Element.div();
}
```

These elements then can be appended to another elements, for example the HTML body or used for other operations like setting their CSS style class and so on.

3.4.7 Mirrors

The mirrors library provide basic reflection for Dart programs with support for *introspection* and *dynamic invocation*. *Introspection* is that subset of reflection by which a running program can examine its own structure. For example, a function that prints out the names of all the members of an arbitrary object. *Dynamic invocation* refers the ability to evaluate code that has not been literally specified at compile time, such as calling a method whose name is

provided as an argument (because it is looked up in a database, or provided interactively by the user).[9]

The Dart mirror system is using mirror objects, e.g. an *InstanceMirror* or *ClassMirror*, that provide an interface to access information about the object and class' structure. A developer can get an *InstanceMirror* object by calling the *reflect* function and passing the object that he wants to get information about. Through the instance mirror, the developer can access its class through a getter call to the *type* member which returns a *ClassMirror* object or an mirror on the field can be obtained and a setter can be invoked. The *ClassMirror* object provides a way how to get a list of fields, methods, getters, setters or constructors or even just a name of the class.

As Smalltalk is a very dynamic system and one can access all the properties, methods in the class object, it is not hard to implement this system. The example code below presents a way how to get an InstanceMirror by calling a *reflect* function and its *ClassMirror* by calling the *type* getter of the *InstanceMirror*. Then we print the name of the class by calling the *simpleName* method which returns an instance of *Symbol* class, which is a part of the *core* library.

```
import 'dart:mirrors';

class MyClass {}

main() {
  var myObj = new MyClass();
  InstanceMirror mirror = reflect(myObj);
  ClassMirror type = mirror.type;
  Symbol name = type.simpleName;
  print(name);
}
```

This prints the following text:

```
Symbol("DartmySrc14::MyClass")
```

It is basically a *toString* method called on the *Symbol* object which represents the *MyClass* symbol. The output is slightly different from the original *Dart* implementation where the output is just "*Symbol("MyClass")*". The difference is in the way how Smalltalk returns the class name. Smalltalk/X return always the namespace(in our case it is an autogenerated library name) and the class name concatenated. It would be possible to get the same output as in the original Dart with a bit of string processing.

The Smalltalk's implementation of the *reflect* function is simply returning the *InstanceMirror* with the the reference to the passed object:

```
reflect:anObject
  ^InstanceMirror InstanceMirror:anObject
```

When we have the *InstanceMirror* for our instance of *MyClass* class, we can ask for the type to get the *ClassMirror* object that holds an instance of the class object which is the *MyClass* class:

```
type
  ^ClassMirror ClassMirror:mirroredObject class
```

Finally, the *ClassMirror*'s simpleName method returns a *Symbol* instance:

```
simpleName
    |symbolName|
    symbolName := DartCore::String
                      withSTString:(mirroredClass
                                        name
                                        asString).
    ^DartCore::Symbol Symbol:symbolName.
```

Then the symbol is simply printed on the output.

3.5 Performance

The performance measurements were run on a MacBook Pro (13-inch, Mid 2012) with 2.5GHz Intel Core i5 and in the OS X El Capitan, version 10.11.6. We have to state here, that the Smalltalk/X version doesn't currently support Just In Time Compilation in its version for OS X and therefore we are comparing just the execution of a the bytecode with the execution in the Dart VM.

The compilation process in Smalltalk is slower as the Dart's compiler and parser are written in C++ and so it is directly executing the machine code without another intermediate level. As our compiler is written in Smalltalk the VM is executing bytecode and this is making it slower. But the interesting part is how fast is our compiled code going to be compared to the execution in the Dart VM.

The performance is compared on a simple bubble sort algorithm whose complexity is $\theta(n)$ where n is the number of elements in the array:

```
sortArray(list) {
  for (int i = 0; i < list.length - 1; i = i + 1) {
    for (int j = 0; j < list.length - 1; j = j + 1) {
      if (list[j] > list[j + 1]) {
        swapWithNext(j, list);
      }
    }
  }
}
```

```
swapWithNext(j, list) {
  var tmp = list[j];
  list[j] = list[j + 1];
  list[j + 1] = tmp;
}
```

We are measuring simply the execution time of the main method which initializes a *List* with 100 numbers and then sorts it with bubble sort algorithm written above. This is done for just one instance to see how much time we need to sort 100 items and we are performing a bigger sort as well which is doing 100 times the same intialize List - sort cycle.

Dart is using *Stopwatch* class for this purpose and we can easily call the *start* method to start the measurement. After our computation is done, we call *stop* and get the elapsed time by accessing the *elapsedMilliseconds* property. For measuring the Smalltalk's execution time, we are using the *TimeDuration* class and its method *toRun:* which accepts directly a block that should be executed and measured. We pass a block with the main method being executed here. The tests were run 10 times and the average execution time is presented in the graphs 3.1 and 3.2.

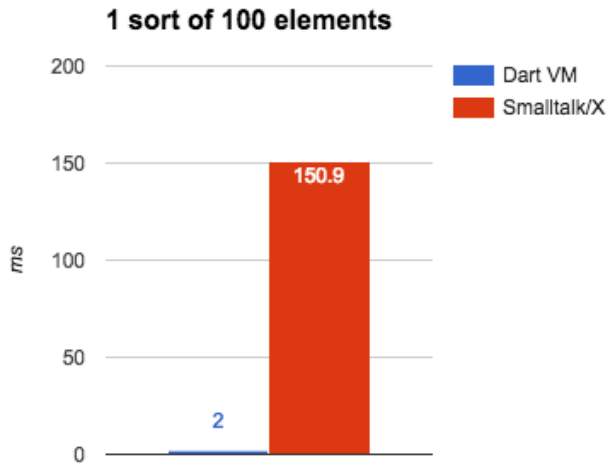


Figure 3.1: Result in ms of measuring one sort of 100 elements

From the results we can clearly see that there is a big lack of performance in our solution compared to the Dart VM. The Dart VM beats the Smalltalk's bytecode execution implementation by more than 100 times.

The JIT compiler can surely improve the performance but due to the issues in OS X version of Smalltalk/X we are not able to measure by how much would JIT compilation improve the performance.

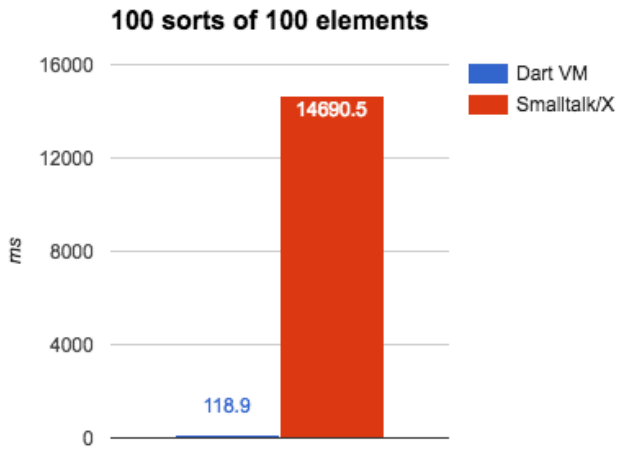


Figure 3.2: Result in ms of measuring 100 sorts of 100 elements

Conclusion

This thesis brought me a deep understanding of how Smalltalk/X's compilation process works, what are the technical details that a usual Smalltalk user or developer doesn't explore. I was also able to examine some very specific parts of the Dart's parser and compiler in detail by reading and studying its implementation.

In the first part I described both Dart and Smalltalk languages and their platforms, summarized the key concepts and features and found the main differences. I have described Smalltalk/X's symbolic code and afterwards I have analysed the options how the Dart code can be compiled into the Smalltalk/X's bytecode. I presented a way how to create equivalent Smalltalk classes, how to compile specific cases and syntactic sugar of Dart and presented selected sequences of compilation into Smalltalk's symbolic code and bytecode.

The implementation is highly inspired by the original open source Dart parser written in C++ but it still keeps the important concept of the symbolic code from the Smalltalk's compiler. Still there are many things to improve and features that are not supported.

In the last part I presented sample codes for Dart native libraries that are supported also in our implementation and I have tested the performance on a bubble sort algorithm which turned out to be much faster in the Dart VM. However, the Smalltalk/X's JIT compiler can bring additional performance improvement which might be more competitive.

Even tough the performance was not matching my initial expectations it was worth it implementing the parser and compiler as I gained experience that helped me understand the architecture of the Smalltalk/X's virtual machine and I hope I will be able to work on this project in the future and make another improvements that will bring competitive results.

Bibliography

[1] Flutter. *Flutter [online]*. [cit. 2017-1-4]. Available from: `https://flutter.io/`

[2] Dart. *Why Not a Bytecode VM? [online]*. [cit. 2016-11-11]. Available from: `https://www.dartlang.org/articles/dart-vm/why-not-bytecode`

[3] eXept Software AG. *Smalltalk/X [online]*. [cit. 2017-1-4]. Available from: `https://www.exept.de/en/smalltalk-x.html`

[4] Bracha, G. *The Dart Programming Language*. ISBN 9780133429954.

[5] Dart. *A Tour of the Dart Language [online]*. [cit. 2016-11-16]. Available from: `https://www.dartlang.org/guides/language/language-tour`

[6] Dart. *Dart API reference documentation, Library dart:core [online]*. [cit. 2016-12-29]. Available from: `https://api.dartlang.org/stable/1.21.0/dart-core/dart-core-library.html`

[7] Dart. *Dart API reference documentation, Library dart:io [online]*. [cit. 2016-12-29]. Available from: `https://api.dartlang.org/stable/1.21.0/dart-io/dart-io-library.html`

[8] Dart. *Dart API reference documentation, Library dart:html [online]*. [cit. 2017-1-3]. Available from: `https://api.dartlang.org/stable/1.21.0/dart-html/dart-html-library.html`

[9] Dart. *Dart API reference documentation, Library dart:mirrors [online]*. [cit. 2017-1-4]. Available from: `https://api.dartlang.org/stable/1.21.0/dart-mirrors/dart-mirrors-library.html`

Appendix A

Acronyms

JIT Just-in-time compiler

VM Virtual Machine

API Application programming interface

URI Unique Resource Identifier

DOM Document Object Model

HTML HyperText Markup Language

CSS Cascading Style Sheets

Appendix B

Contents of enclosed flash drive

- `readme.txt` ..... the file with the contents description
- `src` ..... the directory of source codes
- `examples` ..... the directory with example source codes
- `thesis` ..... the directory of LaTeX source codes of the thesis
  - `DP_Havrila_Branislav_2017.pdf` ..... the thesis text in PDF format